Optimizing the Implementation of Real-Time Simulink Models onto Distributed Automotive Architectures

Gang Han[a], Marco Di Natale[b], Haibo Zeng[c], Xue Liu[c], Wenhua Dou[a]

[a]*National University of Defense Technology, Changsha, China*
[b]*Scuola Superiore S. Anna, Pisa, Italy*
[c]*McGill University, Montreal, Canada*

Abstract

Future automobiles will support an increasing number of complex, distributed functions such as active safety and X-by-wire. In a model-based design flow, system properties can be verified in advance on function models, by simulation or model checking. To ensure that the properties still hold for the final deployed system, the implementation into software tasks and communication messages should preserve the semantics properties of the model. FlexRay offers deterministic communication and can be used to provide distributed implementations that are provably equivalent to models like those created from Simulink, by designing the schedule to ensure the preservation of communication flows. In some cases, such a schedule is not feasible and the model should be modified by adding communication delays. We provide a formulation of the FlexRay scheduling problem that computes the optimal solution with respect to the number of additional delays when a flow-preserving implementation is not possible. The aforementioned scheduling options are applied to an X-by-wire system and a case study with active-safety functions to highlight tradeoffs between schedulability and additional functional delays.

Keywords: Design Optimization, Model-based Design, Real-Time Systems, Time-Triggered Networks, FlexRay

1. Introduction

Automotive electronics and software systems provide an increasing number of complex, distributed and interdependent functions, from the set of active safety features to the X-by-wire functions. These functions share a set of common sensor devices and preprocessing stages (including sensor fusion and object detection) and drive a set of actuators (steer, brake, suspensions, throttle) often under the supervision of arbitration functions. The functions and the controls can be refined in a top-down fashion or constructed bottom-up from component models. To ease the daunting task of developing correct functions with a fast time-to-market, the functions are developed and validated using models and simulation tools like Simulink from The MathWorks. The automotive industry together with the avionic industry was the first to embrace model-based design to simulate the system functions, verify some properties of interest, remove coding errors, and speed up the software development process.

In a model-based design flow, correctness of systems can be demonstrated by formal reasoning upon the models and their desired properties. When exhaustive proof of correctness cannot be achieved, the modeling language should support simulation and automated testing. Formal methods can be used to guide the generation of the test suite and guarantee some degree of coverage. All functional and non-functional constraints and properties that are captured by the system-level and component-level models must be propagated at each refinement step and, in particular, when providing an implementation in software tasks and communication messages. When a software implementation is generated, the designer should have control over the conditions that guarantee the preservation of the model semantics into its refinement. In addition, he/she should be aware of cases in which a formally correct implementation is not possible because of scheduling infeasibility and define a new model that can be feasibly scheduled.

FlexRay [1] is an automotive network communications protocol. It provides support for the transmission of time-critical periodic messages in a static segment and priority-based scheduling of event-triggered messages in a dynamic segment. It offers the possibility of a deterministic communication and can be used to define distributed implementations that are provably equivalent to synchronous reactive models like those created from Simulink. The bandwidth of FlexRay is assigned according to a time-triggered pattern. Clock synchronization, embedded in the standard, ensures deterministic communication at no additional cost. However, the low level communication layers and the FlexRay schedule must be carefully designed to ensure semantics preservation, especially when models include subsystems executing at different rates. In this paper, *we focus on the static segment of FlexRay* protocol, because it provides a deterministic communication and is most likely to be used in the future to support safety-critical applications [2].

1.1. Our Contributions

In this paper, we present possible solutions to the problem of defining FlexRay schedules that preserve the semantics of synchronous reactive models including those generated using

Simulink and Stateflow. When the model behavior cannot be preserved, the schedule can be defined to add known and deterministic delays to the communication links. In this case, the designer may trade performance for schedulability. We estimate the impact of communication delays on feedback control performances by simulation, and define an optimization procedure that tries to find the schedulable solution with the minimum performance loss because of added delays.

To the best of our knowledge, none of the existing work on the scheduling optimization of FlexRay-based automotive systems (e.g. [3] and [4]) considers the requirement of preserving the semantics of synchronous reactive models. We formulate the problem in Mixed-Integer Linear Programming (MILP) framework based on the previous work [3]. Because of the complexity of the problem (formally, NP-completeness has been demonstrated for a subset of our problem, the packing of signals into slots [5]), the one-step MILP formulation cannot handle very large systems. We provide divide-and-conquer techniques to divide the problems in two subproblems of manageable sizes and solve the sub-problems in cascade. We apply the proposed two-step approach to two industrial case studies consisting of a set of active safety functions and an X-by-wire system respectively to demonstrate its effectiveness and efficiency.

The use of MILP (or in general mathematical programming) has both advantages and possible disadvantages. The advantages are the formulation in terms of a set of constraints and metric function is mathematically (and unambiguously) defined, and can be processed by an optimization solver leveraging theory and implementation efforts that have been dedicated for years to this field. Commercial solvers (such as CPLEX) are very efficient and optimized for the purpose of dealing with this type of problems. Second, an MILP formulation can easily accommodate additional constraints to integrate the fact that parts of the design are inherited as legacy or cannot be modified. Finally, MILP solvers work incrementally in the solution of mixed-integer problems and return a sequence of solutions with increasing quality. Thanks to the duality theorem, at any point in time the solver provides not only the current solution, but also an upper bound estimate of the distance to the true optimum. The designer can stop the process at any time, when a solution of sufficient quality is obtained. These advantages come at some price (compared with heuristic search methods or stochastic optimization). First, the problem constraints and the metric function must be expressed such that the feasibility region and the metric function are linear (or convex in the case of convex optimization). In addition, in multi-criteria optimization, the multiple optimization goals have to be combined into a single linear function, with possibly less freedom than the solution evaluation performed in a heuristic search. In our case, an MILP solution was selected because the optimization goal is related to a single aspect (the possible addition of delays) and the linear formulation of constraints and the optimization metrics does not pose significant challenges. This, of course may not be true for other problems.

The rest of the paper is organized as follows. Section 2 summarizes the related work. Section 3 provides an introduction to synchronous reactive models of computation and Simulink. Section 4 defines the FlexRay scheduling problem and the following Section 5 discusses the consequences of scheduling decisions on the functional behavior of the system. The formalization of an optimization problem that tries to find the schedulable solution with minimum delays is presented in Section 6. In Section 7, different scheduling options with deterministic communication delays are then examined for case studies consisting of an X-by-wire and an active-safety automotive system. The paper is concluded in Section 8.

2. Related Work

Synchronous Reactive (SR) models [6] are used for modeling control-dominated embedded applications. SR models are characterized by the "synchronous assumption" or "logical time execution", which requires that the system completes the reaction to an event before the occurrence of any other event. Synchronous languages are implemented in the SCADE commercial tool [7], as well as the very popular MATLAB/Simulink [8] tool chain from The MathWorks. Both commercial toolsets (SCADE and MATLAB/Simulink) allow modeling and simulation of the system according to a synchronous reactive Model of Computation (MoC). In addition, they offer automatic code generators and an interface to verification tools such as the plug-in from Prover [9] for SCADE and Design Verifier for Simulink (also based on Prover technology).

Several research papers have defined a possible formal approach to the problem of semantics preservation upon mapping of function onto architecture, at least in the case where the function is modeled using synchronous reactive MoC. Synchronous models are based on the assumption that the system reaction to any event completes before the next event arrives. In some cases, the mapping of functions onto an architecture may require preservation of this property. Another (relaxed) property of the functional model that can be preserved by its implementation is *flow preservation*, that is, guaranteeing that the implementation operates on the same values of the input data streams as the model. In both cases, the simplest solution is to restrict the functional model to react to periodic events only and to select for its implementation time-triggered execution platforms. This approach is supported by the Time Triggered Architecture [10]. Techniques for generating semantics-preserving implementations of synchronous models on TTA have been studied in [11]. In [12], an initial discussion is provided for the possible tradeoffs when defining a FlexRay communication schedule in a model-based design flow.

Methods for desynchronization in distributed implementations have been studied and presented in [13, 14]. A more general approach consists of an intermediate mapping of synchronous models into Kahn Process Networks [15], for which a correct implementation in an unsynchronized architecture platform can be found more easily [16] (albeit, very likely at the price of additional overhead and pessimism in the time analysis).

There is also a large body of work in the use of other models of computation for the design and optimization of embedded

systems. An overview of possible models of computation, analysis and synthesis techniques can be found in [17] and [18].

There is abundant literature on code synthesis techniques for other synchronous languages, including Esterel [19], Lustre [20], and Prelude [21]. The generation of a single-task software implementation for Esterel and Lustre models is discussed in [22] [23] [24]. Reactions decompose into atomic actions that are partially ordered by the causality analysis of the program. The scheduling is generated at compile time trying to exploit the partial causality order of functions and the generated code executes without the need of an operating system. The generated code is optimized according to code size, speed and efficiency of the compilation process.

A different model of computation is synchronous dataflows (SDF [25]), in which the advancement of computation relates to the productions/availability of (data) tokens (not necessarily associated with a timed event). Code generation techniques for SDFs are discussed in [26], [27] and [28]. More recent developments include clustering techniques for implementation on multiprocessors-on-chip and GPUs [29]. Contrary to this work, in the synthesis of a code implementation for synchronous dataflows, latencies (and real-time schedulability) are not the primary concern. The main objectives are the optimization of the processing rates and the minimization of the computation buffers.

Other formal models for which code generation techniques are available include heterogeneous representations such as FunState [30]. In this case, system optimization is performed across the interface/behavior language boundaries. Finally, the concurrent and synchronous (rendezvous based) model of Hoares communicating sequential processes (CSP) [31] and the pure event-based model of UML Statecharts [32] are other models to which automatic software synthesis techniques are successfully applied [33].

Techniques for optimization include search-based heuristics [26], graph clustering and reduction techniques [34] [29] and MILP optimization methods [35]. A hybrid combination of genetic algorithm and ant colony optimization is used in [36] for allocating and scheduling tasks. In [37] an automatic platform-based system synthesis procedure based on the technique of Satisfiability Modulo Theories is proposed, which can check the system feasibility with respect to functional and nonfunctional constraints. MILP optimization techniques are also used for similar synthesis problems driven by performance optimization, including the problem of HW/SW partitioning [38].

However, none of these papers considers jointly the problem of controls performance and buffer optimization under (distributed) real-time schedulability constraints derived from the semantics preservation of synchronous reactive models. This is also the main difference with our previous work [3], in which only schedulability of the FlexRay bus was considered (without consideration of semantics preservation issues).

Scheduling techniques for the FlexRay static segment have been developed by extending the work for scheduling messages in a TDMA bus [39, 40]. In [39], the authors present a static cyclic scheduling technique for time-triggered messages. In [5] both a fast heuristic and a Mixed-Integer Linear Programming (MILP) optimization formulation are proposed for the problem of PDU (Protocol Data Unit) to message packing. In [41], in addition to the minimization of the number of used slots, the authors also present a formulation for the minimization of the transmission jitter. [42, 43] propose to use message retransmissions in the FlexRay static segment to provide guarantees on reliability. In [44], the authors consider the case of a hard real-time application implemented on a FlexRay system. Messages are scheduled in the static segment only. In [45], the authors present timing analysis of applications communicating over FlexRay, for both the static and the dynamic segments. The authors first present a static cyclic scheduling technique for messages transmitted in the static segment. Then, they develop a worst-case response time analysis for event-based transmissions in the dynamic segment. Message analysis is integrated in a holistic method that computes the worst-case response times of all tasks and messages. In [2] the authors discuss a system-level design optimization problem. However, they assume a communication model where task and message schedules are not synchronized, and the problem considered in the paper is characterized by data freshness constraints only. [3] uses MILP to address the scheduling synthesis for FlexRay-based systems that are subject to timing constraints such as latency and extensibility with a synchronous communication model. A similar approach is taken in [4], but the scheduling is done at the task-level (as opposite to job-level in [3]). However, neither [3] nor [4] accounts for flow preservation of synchronous reactive models.

Few solutions exist for the design optimization problem. Starting from [46], the authors discuss the use of genetic algorithms for optimizing allocation and priority assignments of real-time tasks with respect to a number of constraints, including end-to-end deadline and jitter. A similar problem is discussed in [47], where an MILP formulation is used to jointly optimize priority assignments and allocations of tasks and messages. The formulation was extended from single-bus systems to systems with gateways in [48]. In [49], a design optimization heuristic for mixed time-triggered and event-triggered systems was proposed. The algorithm assumes that nodes are synchronized, with an architecture similar to the one discussed in this work. An algorithm for the optimal synthesis of task priorities for distributed systems with time-triggered and priority-based scheduling, based on the performance of the controls is presented in [50]. In [51], a SAT-based approach was proposed for the placement and priority assignment problem. In [52], task allocation and priority assignment were defined with the purpose of optimizing extensibility with respect to changes in task computation times. The proposed solution was based on simulated annealing on a parallel computing cluster. In [53], a generalized definition of extensibility on multiple dimensions was presented and a randomized optimization procedure based on a genetic algorithm was proposed to solve the optimization problem. Again, none of these design optimization methods deals with the problem of flow preservation in a model-based development process, in particular, with the constraints and the scheduling tradeoffs arising from the consideration of preserving the semantics of a Simulink model.

3. Synchronous Models and Simulink

In Simulink, the system is defined as a network of communicating blocks. Each block operates on a set of input signals and produces a set of output signals, according to its specifications. Formally, a block transforms input functions (of time) into output functions. The input function domain can be a set of discrete points (discrete-time signal) or can be defined on a continuous time interval (continuous-time signal). During code generation, continuous blocks are implemented by a fixed-step solver, executing at the base rate. Eventually, every block has a sampling rate, with the restriction that the discrete part is executed at the same rate or at an integer fraction of the base rate.

Simulink computes for each block, at each step, the set of outputs, as a function of the current inputs and the block state, and then, it updates the block state. A cyclic dependency among blocks where output values are instantaneously produced based on the inputs results in a fixed point problem and possibly inconsistency. A fundamental part of the model semantics is the rules dictating the evaluation order of the blocks. Any block whose output is directly dependent on its input (i.e., any block with *direct feedthrough*) cannot execute until the block driving its input has executed. Some blocks set their outputs based on the values of state variables, updated independently from the inputs. The set of topological dependencies implied by direct feedthrough blocks defines a partial order expressed as a set of precedence constraints among pairs of blocks. The partial order must be accounted for in the simulation and correct implementation of the model.

Before Simulink simulates a model, it orders all blocks based upon their topological dependencies. This includes expanding subsystems into the individual blocks they contain and flattening the entire model into a single list. The tool chooses one total order in the execution of blocks that is compatible with the partial order imposed by the model semantics. Next, the virtual time is initialized at zero, the simulator engine scans the precedence list, and executes the blocks for which the value of the virtual time is an integer multiple of the period of their inputs.

Executing a block means computing the output function, followed by the state update function. When the execution of all the blocks that need to be triggered at the current instant of the virtual time is completed, the simulator advances the virtual clock by one base rate cycle and resumes scanning the block list. In an example multi-rate system, represented in the left side of Figure 1 (case (a)), characterized by oversampling of the communication, a possible order of execution of the blocks at simulation time would be the one represented in the upper timeline (labeled as (a), bottom part of the figure). Block C is executed at the base rate and, because of the feedthrough dependencies, it must follow both A and B. However, a different execution order could be obtained by executing block C first. This execution order corresponds to the model on the right side, labeled as (b), in which a delay of one time unit is added to the communications from A and, respectively, B to C.

Such added delays change the behavior of the model and affect the performance of the control algorithm. We use a relatively complex example taken from the library of the Simulink

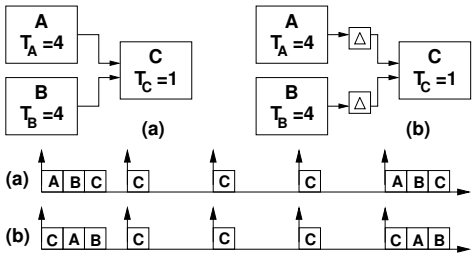


Figure 1: An example of simulation-time execution order.

tool to illustrate the impact of added delays. Figure 2 shows a Simulink model of a hydraulic servomechanism controlled by a pulse-width modulated (PWM) solenoid. It is a representative of feedback control loops in which there is a flow of data from the sensor to the control and from the control back to the actuator. If we assume the system is implemented in a distributed platform, and data communications between the sensor, the actuator, and the control occur over a FlexRay bus (in the figure, this is represented by the dashed rectangle over the communication links), we can simulate the effect of added delays.

Figure 3 shows the same hydraulic servo model, with additional delay blocks modeling the effect of communication delays of 2 and 3 FlexRay cycles (with a cycle time of 5ms) on the sensor and actuator paths.

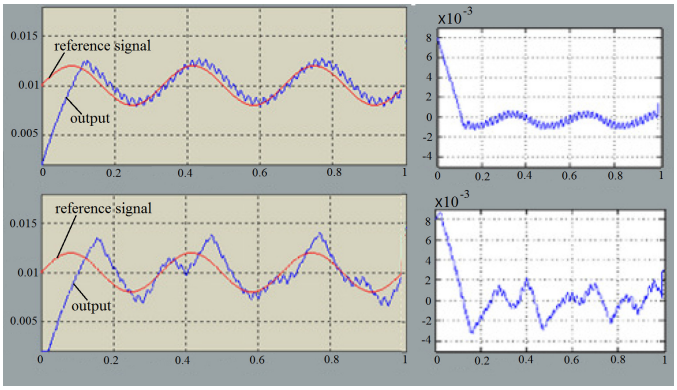


Figure 4: Actuator position and error for the hydraulic servo without (top) and with (bottom) delays.

Figure 4 shows the simulation results of the example model, in the top row without added delays, and in the bottom row when a unit delay is added on the sensor path. The figure shows the reference signal (left graph), the output (left graph), and the error (right graph) in these two cases. The control quality with a unit delay is somewhat degraded, and the simulation results show an error about four times larger than the one without delay. For this control model, it is possible to measure the control error on the given reference signal for different delay (Δ) values on the actuator and sensor paths. The results are shown in Table 1.

In this case, if the control error is the performance parameter of interest, it is possible to associate to each delay value a performance cost. If convex optimization is used to compute an optimal design configuration, we will need to approximate

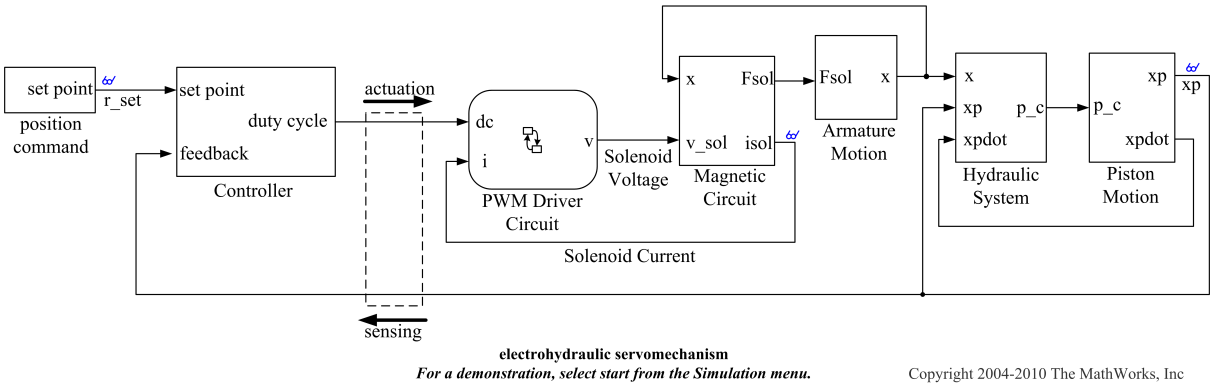


Figure 2: A Simulink example of an hydraulic servomechanism (representative of a suspension control).

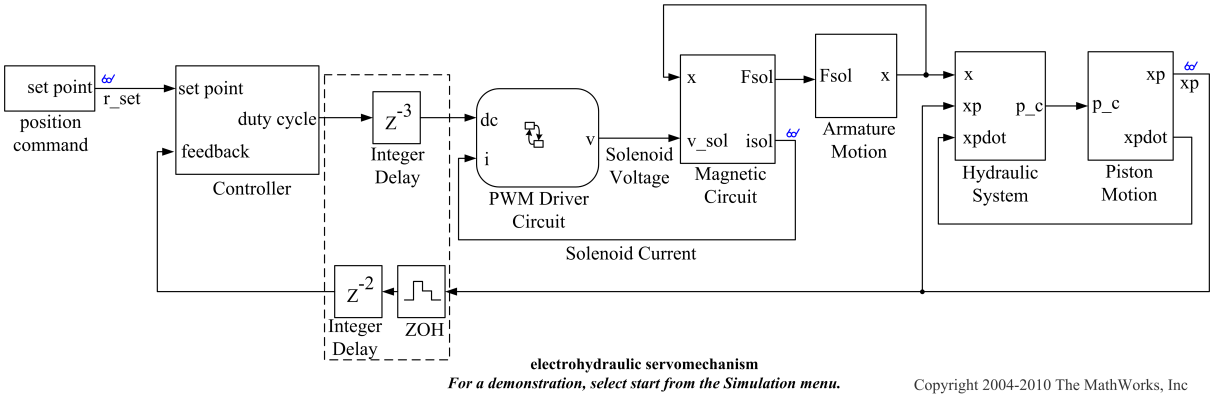


Figure 3: Hydraulic servo with additional communication delays.

sensor Δ	actuator Δ	max error (mm)	sensor Δ	actuator Δ	max error (mm)
0	0	0.75	0	2	1.8
1	0	1.1	1	2	2.25
2	0	1.25	2	2	2.5
3	0	1.75	3	2	2.8
0	1	1.6	0	3	1.8
1	1	1.6	1	3	2.25
2	1	2.2	2	3	2.5
3	1	2.8	3	3	2.8

Table 1: Max errors for different delays (in 5ms units) on the sensor and actuator paths

the dependency of the performance from the number of delays with a convex function. For example, we can find a convex hull or even linearize the function expressed by the table using a least square approximation. In our example, a least square approximation returns the two linear coefficients $\beta_1 = 0.3895$, $\beta_2 = 0.4095$ such that the max error e can be approximated as a function of the number of sensors and actuator delays by $e = 0.75 + \beta_1\Delta_s + \beta_2\Delta_a$ with an average linearization error of 10%. In many cases, it is typically accurate enough to use continuous piecewise linear function to approximate such curves, and use such functions in MILP [54].

Of course, using simulation to find weights to the links possibly affected by delays may be a time-consuming task for large size systems. First, it should be considered that, at least for the purpose of this work, only the links/signals mapped into FlexRay communication needs to be considered by the analysis. This is typically a small subset of all the links in the system. For each control loop or path on which n links can be affected by FlexRay communication delays of at most Δ units, $(\Delta + 1)^n$ runs are required. Even if loops could be analyzed assuming no cross dependencies and the number of links affected for each loop is typically small, designers may want to limit the analysis only to those loops that they estimate are most sensitive to the added delays.

In the following sections, we discuss the FlexRay scheduling problem, highlight the possible tradeoffs when defining a FlexRay communication schedule in a model-based design flow, and then present an MILP formulation.

4. Task and Message Scheduling in FlexRay-based Systems

FlexRay is a modern communication standard for highly deterministic and high speed communication [1]. In FlexRay, the maximum communication speed is defined at 10 Mb/s, and the bus bandwidth is assigned according to a time-triggered pattern. The available bandwidth is divided in *FlexRay communication cycles* with equal length, and each cycle contains up to four segments (Static, Dynamic, Symbol and Network Idle Time - NIT). Clock synchronization is embedded in the standard, using part of the NIT segment (Figure 5).

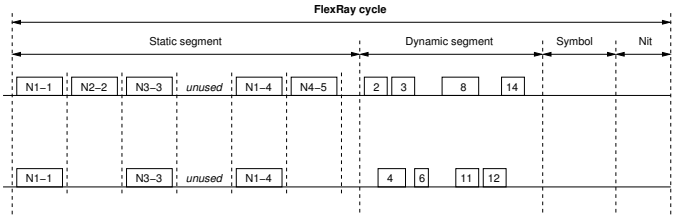


Figure 5: The FlexRay communication cycle and its four segments.

The static segment of the communication cycle enables the transmission of time critical messages according to a periodic pattern or schedule, in which a time *slot*, of fixed length and in a given position in the cycle, is always reserved to the same node. In the latest version 3.0.1 of FlexRay [1], slot multiplexing is allowed in the static segments, i.e., slots with the same index in different communication cycles can be owned by different Electronic Control Units (ECUs). Each node only needs to know the time slots for its outgoing and incoming communications. The specification of these time slots is kept in local scheduling tables. As long as the local tables are consistent, no timing conflicts or interferences arise. Slots that are left free in the (virtual) global table resulting from the composition of the local tables can be used for future extensions. Time protection and isolation from timing faults are guaranteed by the reservation of time slots and guardians that avoid node transmitting outside the allocated time window.

In our study, we are interested in the flow-preserving implementation of a model consisting of computations (the block functions) and communications (the signal links). Hence, our model of the implementation must include tasks and messages and consider the integrated scheduling of all of them. We consider a model of the system computations as a *dataflow graph* V. The vertices represent the *tasks* and the edges represent the data *signals* communicated among them.

A task τ_i is characterized by the tuple $(e_i, T_i, \Phi_i, J_i, C_i, d_i)$, where e_i is the ECU resource it needs to execute, T_i is its period, Φ_i is its initial phase, J_i is its release jitter, C_i is its execution time, and $d_i \leq T_i$ is its deadline. For priority-based scheduling systems such as OSEK [55], each task τ_i is assigned with a static priority P_i, and the scheduling policy is assumed to be fully-preemptive. We use the convention in OSEK: the higher the number is, the lower the priority level is, thus $P_i < P_j$ implies that τ_i has a higher priority than τ_j. We also denote the set of tasks with higher priority than τ_i and executed on the same ECU as $hp(i) = \{j : e_j = e_i, P_j < P_i\}$.

Edges represents the input/output connections between tasks. An edge between tasks τ_h and τ_k denote a data signal $\sigma_{h,k}$ with a given bit width $b_{h,k}$ produced by τ_h and available to τ_k. We follow the requirement from Simulink that the data communication only happens between tasks with harmonic periods, i.e. either $T_h = j \times T_k$ or $T_k = j \times T_h$ for some positive integer j. For simplicity, signals will also be identified and denoted by a single index as in σ_i. Each periodic task reads its input at its activation time and writes its results at the end of its execution. Each signal σ_i may optionally be delivered with a Δ_i-*unit delay*, e.g. the signals from τ_8 to τ_2 and τ_6 in Figure 6 carry a one-unit delay. Each signal also carries a precedence constraint in the execution of the sender and receiver job. If the signal is delivered without delay, the successor must be executed after the sender job instance activated immediately before it, but before the following one; otherwise, it is the Δ_i-th job of the successor which will use the signal value produced by the sender (see Figure 7). To quantify its importance to the system control performance, we assign a weight $w_i \geq 0$ to each signal σ_i.

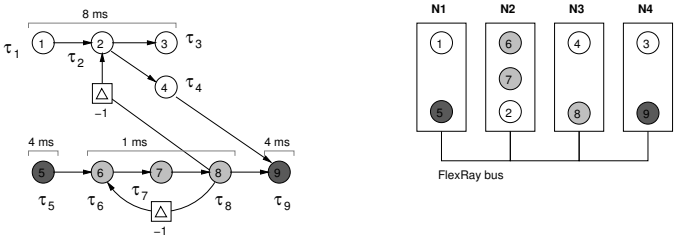


Figure 6: A task model with a unit-delay communication and allocation of tasks.

The hyperperiod or *application cycle* H is defined as the least common multiple (lcm) of the periods of all tasks. Inside the application cycle, each job is considered as an individual scheduling entity. The scheduling problem consists of planning the execution of jobs and the transmission of signals into the available slots inside H. Jobs can also be denoted with reference to their task. In this case, $t_{k,j}$ denotes the j-th job of task τ_k.

The *arrival time* of a job instance t_i is denoted as a_i or, using the instance index notation as $a_{k,j}$, with $a_{k,j} = \Phi_k + (j-1) \times T_k$. It indicates the time instant when the job is signalled to be available for execution. The *release time* of a job is A_i, the time instant when the job is actually ready for execution. The *finishing time* is f_i. The *response time* r_i of a job t_i is the time interval from its arrival to its termination, i.e. $r_i = f_i - a_i$. The worst-case task response time R_k of task τ_k is the maximum of the response times $r_{k,j}$ of its jobs $t_{k,j}$. The worst-case jitter J_k of task τ_k is the maximum difference between the arrival time and release time of all the jobs of τ_k (typically representing activation delays because of interrupt response times and the execution time of the scheduler itself).

The set of all the task instances transmitted in the application cycle defines the application instance graph, as in Figure 7. The FlexRay communication stack may transmit multiple signals in the data content of a single message in a communication slot, upon conditions that they are transmitted by tasks allocated to the same node.

A *path* from τ_i to τ_j, or $P_{i,j}$, is a sequence $p = [\tau_i, \ldots, \tau_j]$ of tasks such that there is a link between any two consecutive tasks. For example, in Figure 6 a path exists between tasks τ_1 and τ_9. The *latency of path* $P_{i,j}$ is defined as the time interval between the arrival of one instance of τ_i and the completion of the instance of τ_j that produces a result dependent on the output of τ_i. The end-to-end latency of a path p should satisfy its deadline requirement D_p.

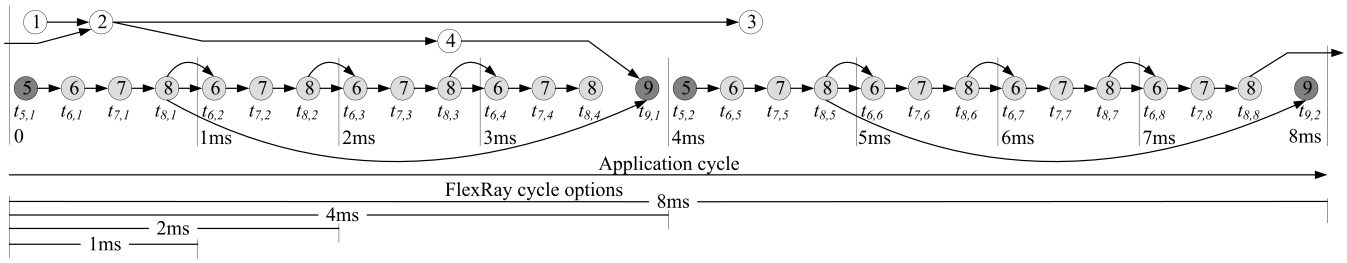


Figure 7: Unrolling the task model in task instances in the application cycle.

The scheduling of FlexRay communication consists of the mapping of the tasks and signals defined in the application cycle into a set of communication cycle instances. This mapping can be performed in different ways, according to the selection of the communication cycle length, the size of the static segment, the slot size and correspondingly the number of static slots in each communication cycle. It is practically impossible to encode all the above into an integrated problem formulation to be solved by an optimization framework without having an exceedingly large search space.

Hence, in our previous research work [3], we investigated a two-step design flow. Starting from the design specification, we assumed a FlexRay bus configuration (l_{app}, l_{comm}, n_{slot}, l_{slot}, b_{slot}) is given, where l_{app} is the length of the application cycle (the least common multiple of the task periods), l_{comm} is the length of the FlexRay communication cycle, n_{slot} is the number of slots in the static segment of the communication cycle, l_{slot} is the length of the slot in time, and b_{slot} is the size of the slot in bits. Based on this configuration, we apply a mathematical programming framework to encode the problem and synthesize other variables such as slot ownership, signal to slot mapping, message and task scheduling. The two-step design flow is consistent with the typical design flows in use by the automotive industry, where the communication cycle and the slot size are defined based on the need to reuse legacy components and standardize configurations.

In [3] we formulated our problem in the general framework of mathematical programming (MP), where the system is represented with parameters, decision variables, and constraints over the parameters and decision variables. An objective function, defined over the same set of variables, characterizes the optimal solution. The FlexRay scheduling problem allows a mixed integer linear programming (MILP) formulation that is amenable to automatic processing.

5. Trading Functional Delays for Schedulability

The example in Figure 1(a) is a case of communication with oversampling. A FlexRay schedule that preserves the execution order defined by the model semantics and used to validate the system behavior at simulation time should execute as in the following Figure 8.

As is clear from the figure, this can be a very tightly constrained scheduling problem. Both senders must be executed before the FlexRay slots allocated for communication of their

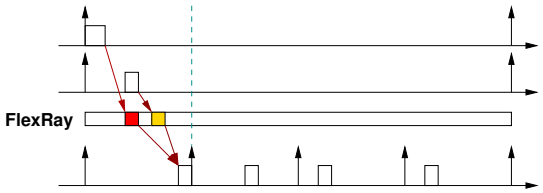


Figure 8: Scheduling the tasks and the communication of the example without delays.

output data. Those slots, in turn, should be allocated so that they precede the scheduling of the receiver task, which needs to complete before the end of its period. In conclusion, the entire chain must be scheduled before the deadline of the receiver task (shown as a dotted line in the figure).

In a development flow, the system designer and the software engineers are presented with a scheduling problem that defines the execution rate constraints and the execution order constraints. In the case of Figure 1, the scheduler would be requested to execute both senders within their period of 4 units and, following the sender task, to assign the communication slots on the bus for transmission. Receivers would be scheduled in such a way that one instance is executed every unit. The requirement that the first instance of the receivers should process the data (in every cycle of 4) should also be considered. However, because of the tight deadlines, the designer could be tempted to release the execution constraints to ease schedulability, by adding functional delays to the communication links, trading functional performance for ease of schedulability.

The scheduling problem may be relaxed if we can select the receiver instance that reads the data produced by the sender. This choice, however, is not neutral to the behavior of the functions that are executing the control algorithms.

Consider the case in which one of the communication paths is delayed allowing communication with the third instance instead of the first one. In this case, one task and one signal can be scheduled later to ease feasibility of the system-level schedule (including the FlexRay scheduling, Figure 9).

An implementation that should be avoided is when even deterministic delays are not guaranteed by the scheduler. If the schedule is generated by only looking at the periods of the communicating tasks, with the guarantee that the execution of each task and the transmission of each signal occur only once during the period, then the situation could be as in Figure 10, where the simulation of the resulting system is more complicated and

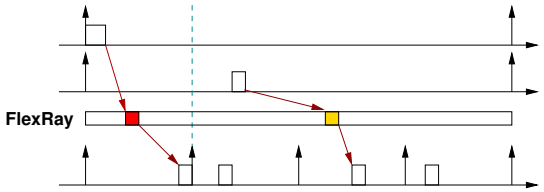


Figure 9: Scheduling the tasks and the communication of the example with a deterministic delay of two receiver periods on one of the communication links.

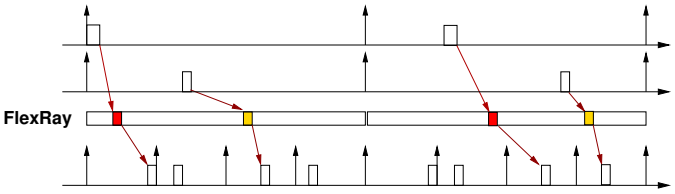


Figure 10: An example of a schedule with nondeterministic delays.

the resulting behavior is less predictable. Such a situation is unfortunately too common.

By formally defining the (real-time) scheduling problem with the flow preservation constraints and the optional addition of delays on communications, it is possible to encode the design problem as a formal MILP problem, in which we seek the feasible solution with minimum performance penalty because of communication delays. An MILP problem can be solved by standard solvers quite efficiently. Compared with possible alternate solutions of developing a heuristic or using a stochastic optimization method, it has the advantage of having the guarantee of optimality when the solver manages to compute the optimum and (especially) an upper bound on the distance of the intermediate solutions to the global optimum.

6. Minimization of Functional Delays

We use an MILP formulation to find a solution to the system scheduling problem (including the FlexRay bus) with respect to a cost function that accounts for the optimization of the system control performance. With respect to the MILP formulation in [3], the main differences are the constraints that depend on the delays of the communication links (Sections 6.2 and 6.4).

6.1. Task scheduling

T_i and d_i denote the period and relative deadline for periodic task τ_i ($d_i \leq T_i$). $a_{i,k}$, $A_{i,k}$, $f_{i,k}$ denote the arrival time, activation time, and finish time for a job $t_{i,k}$, where k represents the job instance index.

T_i and d_i are input parameters, while Φ_i, and consequently $a_{i,k}$, $A_{i,k}$, $f_{i,k}$ are design variables for the optimization framework. Given that all tasks are periodic with an initial phase, the arrival times of the jobs must be constrained accordingly.

$$\begin{aligned} \forall t_{i,k}, a_{i,k} &= \Phi_i + (k-1)T_i \\ 0 &\leq \Phi_i < T_i \end{aligned} \tag{1}$$

$A_{i,k}$ is linked to $a_{i,k}$ through J_i. $a_{i,k}$ is the "ideal" periodic activation time, as provided by a hardware interrupt coming from a clock. $A_{i,k}$ is the corresponding time when the periodic task (that should arrive at $a_{i,k}$) is released into the system, or more precisely, put into the ready queue by the interrupt handler routine. In OSEK [55], tasks are activated periodically, by an internal dispatcher or by an alarm and scheduled according to their priorities. As previously stated, the response time of the scheduler may introduce jitter in the activation time. In OSEK systems, we expect the activation jitter to be significant. Therefore, we include a jitter term J_i in our formulation.

$$\forall t_{i,k}, A_{i,k} - a_{i,k} = J_i \tag{2}$$

The worst-case response time R_i is computed for the tasks and applies to all their jobs (where $hp(i)$ is the set of tasks with priority higher than τ_i).

$$R_i = J_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i - J_i + J_j}{T_j} \right\rceil C_j \tag{3}$$

Since all the parameters in Equation (3) except R_i are known, R_i can be computed beforehand and used as a parameter. Real-time schedulability theory tells us that the value of R_i computed according to Equation (3) is the largest possible for any possible offset assignment and it is therefore safe to use (but pessimistic) for all the jobs of τ_i for the purpose of defining end-to-end deadline constraints and latency metrics. Hence, the upper bounds on the finish times of jobs also exhibit a periodic pattern.

$$\forall t_{i,k}, f_{i,k} = a_{i,k} + R_i = \Phi_i + (k-1)T_i + R_i \tag{4}$$

6.2. Data dependencies

The constraints in this section are specific to the problem of providing a flow-preserving implementation with the possible addition of delays.

The amount of possible delay (an integer variable Δ_i) on a communication link defines the instances of the sender and receiver jobs that are affected by an order of execution (precedence) constraint. Consider a (remote) communication signal σ_i from τ_h to τ_k. If the communication of σ_i is associated with a variable number of unit delays equal to Δ_i, then the j-th instance of τ_h, produces data that is consumed by the instance of τ_k defined by the (variable) index $n_{k,h,j}$ according to the relation ($n_{k,h,j}$ depends on h and j and is the index of the job following $a_{h,j}$ with a further delay of Δ_i).

$$a_{k,n'-1} < a_{h,j} \leq a_{k,n'}, \text{ where } n' = n_{k,h,j} - \Delta_i \tag{5}$$

As both $n_{k,h,j}$ and Δ_i are integer variables in the formulation, we transform the above inequality to an MILP constraint

$$\begin{aligned} \Phi_h + (j-1)T_h &> \Phi_k + (n_{k,h,j} - \Delta_i - 2)T_k \\ \Phi_h + (j-1)T_h &\leq \Phi_k + (n_{k,h,j} - \Delta_i - 1)T_k \end{aligned} \tag{6}$$

We impose the data dependency between the j-th job of τ_h and the $n_{k,h,j}$-th job of τ_k, by making sure that $t_{h,j}$ finishes execution before σ_i is scheduled for transmission on the bus, and σ_i finishes its transmission before the receiver job $t_{k,n_{k,h,j}}$ arrives.

$$f_{h,j} \leq s_i \Rightarrow \Phi_h + (j-1)T_h + R_h \leq s_i \tag{7}$$

$$f_i \leq a_{k,n_{k,h,j}} \Rightarrow f_i \leq \Phi_k + (n_{k,h,j} - 1)T_k \tag{8}$$

Also, to avoid that the data in signal σ_i is overwritten by the next job ($t_{h,j+1}$) of the same sender task, σ_i needs to start transmission before the arrival of $t_{h,j+1}$, i.e.,

$$s_i \leq a_{h,j+1} \tag{9}$$

If the communication between jobs t_i and t_l occurs on the same ECU, we can assume the existence of flow-preserving wait free communication buffers (for a description please refer to [56]) that guarantee a correct implementation upon condition that t_i is activated before t_l (or the sender finishes execution before the arrival of the receiver if it has lower priority).

$$\begin{cases} A_i \leq a_l, & \text{if } P_i < P_l \\ f_i \leq a_l, & \text{if } P_i > P_l \end{cases} \tag{10}$$

6.3. FlexRay protocol rules

This section provides a summary of the FlexRay scheduling problem constraints. The detailed MILP formulation for our method can be found in [3]. In a communication cycle, $s^s_{j,k}$ is used as an input parameter which denotes the starting time of the k^{th} slot from the j^{th} communication cycle. $s^s_{j,k}$ is easily calculated as $s^s_{j,k} = j \cdot l_{\text{comm}} + k \cdot l_{\text{slot}}$.

The mapping of signals to slots is encoded in a set of binary variables

$$A_{i,j,k} = \begin{cases} 1, & \text{if } \sigma_i \text{ is mapped to cycle } j, \text{ slot } k \\ 0, & \text{otherwise} \end{cases} \tag{11}$$

If a signal σ_i is mapped to the k^{th} slot of the j^{th} cycle, the start time s_i and finish time f_i of σ_i are automatically constrained to the start time $s^s_{j,k}$ and finish time $f^s_{j,k}$ of the slot. This can be formulated using the standard "big-M" formulation (where M is a constant larger than any of the variables in the constraint)

$$s^s_{j,k} \leq s_i + (1 - A_{i,j,k})M \tag{12}$$

$$s_i \leq s^s_{j,k} + (1 - A_{i,j,k})M \tag{13}$$

$$f_i = s_i + l_{\text{slot}} \tag{14}$$

Each signal can only be mapped to one slot and the sum of the payloads over all the signals mapped into a specific slot will be upper bounded by the slot size.

$$\sum_{s^s_{j,k} \leq t_{\max}} A_{i,j,k} = 1 \tag{15}$$

$$\sum_{i \in \text{Signals}} A_{i,j,k} \cdot b_i \leq b_{\text{slot}} \tag{16}$$

where $t_{\max}$ is the maximum time span over which the planner must compute a schedule (see Section 6.5).

Each slot is owned by an ECU or it is free. A set of binary variable encodes the status of each slot

$$A_{e_i,j,k} = \begin{cases} 1, & \text{if slot } k \text{ in cycle } j \text{ is owned by ECU } e_i \\ 0, & \text{otherwise} \end{cases} \tag{17}$$

FlexRay has its requirement for the slot ownership. The new FlexRay standard allows slot multiplexing, such that the same slot index at different communication cycles can be allocated to different ECUs. Constraint (18) encodes slot ownership. If signal σ_i is mapped to communication cycle j and slot k, then its source ECU must own slot k. (19) ensures that every slot is owned by at most one ECU. In the latest FlexRay standard 3.0 [1], slot multiplexing is allowed, i.e., slots with the same index in different communication cycles can be owned by different ECUs. If no signal is mapped to slot k in communication cycle j, then constraint (20) sets the slot ownership to null.

$$A_{i,j,k} \leq A_{e_i,j,k} \tag{18}$$

$$\sum_{e_p \in \text{ECUs}} A_{e_p,j,k} \leq 1 \tag{19}$$

$$A_{e_p,j,k} \leq \sum_{i \in \text{signals}, j < n_a} A_{i,j,k} \tag{20}$$

where n_a is the number of communication cycles in $[0, t_{\max}]$.

6.4. End-to-end latency

If Π is the set of latency-sensitive paths, we impose that the end-to-end delays of the paths are within their deadline constraints. However, for a path $p = [\tau_1, \cdots, \tau_k]$, given the instance n_1 of the sender task, *the sink instance varies depending on the delay of the communication links*. We use the following set of constraints to identify the index of the sink, and consequently the end-to-end latency.

$$\begin{aligned} &\forall (\tau_i, \tau_{i+1}) \in p, \forall n_1, \\ &\Phi_{i+1} + (n_{i+1} - \Delta_{i,i+1} - 1)T_{i+1} \geq \Phi_i + (n_i - 1)T_i \end{aligned} \tag{21}$$

where $\Delta_{i,i+1}$ denotes the delay in the communication link (τ_i, τ_{i+1}). Now the end-to-end latency can be formulated as

$$\begin{aligned} & f_{k,n_k} - a_{1,n_1} \leq D_p \\ \Rightarrow \quad & (\Phi_k + (n_k - 1)T_k + R_k) - (\Phi_1 + (n_1 - 1)T_1) \leq D_p \end{aligned} \tag{22}$$

6.5. Scheduling window

If Φ_i is the initial phase of a generic task τ_i, the scheduling of the tasks and of the FlexRay bus must be performed until an entire application cycle has been computed. This means that the schedule must continue until time $H + \max_i(\Phi_i)$. Since the initial phase values are computed as a result of the optimization, we will use an upper bound for the previous formula

$$t_{\max} = H + \max_i(T_i) \tag{23}$$

In the interval $[0, t_{\max}]$ (see Figure 11) we need to schedule for each task τ_i a number of instances

$$n_i = \left\lceil \frac{t_{\max}}{T_i} \right\rceil \tag{24}$$

However, not all of those instances can be scheduled freely. In the example of Figure 11, this is true for $t_{3,3}$, but not for $t_{1,7}$, which must be scheduled in a position defined by $t_{1,1}$ because

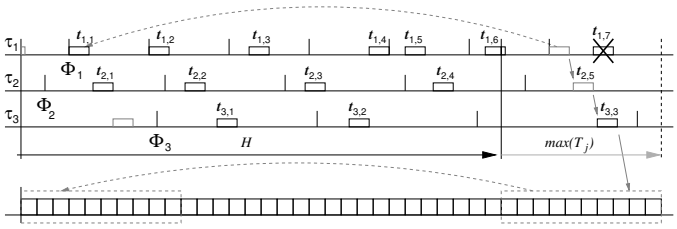


Figure 11: Periodicity constraints in the definition of the scheduling table

they are actually the same instance in the hyperperiod cycle. This translates into the constraint on the finish times for both types of schedulers.

$$f_{i,q} = f_{i,k} + H \text{ where } q = nc_i + k, \tag{25}$$

where nc_i is the number of jobs in one hyperperiod for task τ_i

$$nc_i = \frac{H}{T_i} \tag{26}$$

Similar constraints exist on the scheduling of the FlexRay slots. We need to schedule beyond the application cycle, up to n_a number of cycles, where

$$n_a = \left\lceil \frac{t_{\max}}{l_{\text{comm}}} \right\rceil \tag{27}$$

in the last cycle, however, only

$$n_l = \left\lfloor \frac{t_{\max} - (n_a - 1) l_{\text{comm}}}{l_{\text{slot}}} \right\rfloor \tag{28}$$

slots need to be considered. Similar to job scheduling, signal to slot mappings must match when they refer to the same position in the application cycle. The matching set of signals can be identified as follows. If the sender and receiver jobs of signals σ_i and σ_j are exactly one application cycle away, i.e.,

$$\begin{array}{ll} & \text{src}(\sigma_i) = t_{k,l} \wedge \text{dst}(\sigma_i) = t_{m,n} \\ \wedge & \text{src}(\sigma_j) = t_{k,p} \wedge \text{dst}(\sigma_j) = t_{m,q} \\ & \text{where } p = l + nc_k \text{ and } q = n + nc_m \end{array} \tag{29}$$

then the two signals are actually the same and must be allocated to the corresponding slots (with a distance of H). We denote this relationship as $\sigma_i = \sigma_j$. For each cycle k and slot index l and, for each pair $\sigma_i = \sigma_j$, it must be

$$A_{j,m,l} = 1 \text{ if and only if } A_{i,k,l} = 1 \wedge m = k + H/l_{\text{comm}} \tag{30}$$

6.6. Objective functions

Subject to the satisfaction of the above constraints, we can seek optimality with respect to different cost functions. In the experiments, we seek to minimize the weighted sum of the functional delays on the communication links.

$$\text{minimize} \sum_{\sigma_i \in \text{Signals}} (w_i \cdot \Delta_i) \tag{31}$$

If the relationship of the control error and the added functional delays is given, we can also use continuous piecewise linear functions to approximate such curves, and minimize the weighted sum of the control errors. Because of the lack of enough information on the Simulink models (and thus the exact quantification of the control errors), we leave such study to future work.

7. Experiments

Our experiments have been performed on sets of tasks and signals extracted from actual automotive systems. The first set of experiments provide a simple example of how the addition of delays allows for a feasible schedule of an otherwise unschedulable system. The following case studies provide examples of the design optimization procedure presented in the previous section.

7.1. Leveraging functional delays to improve schedulability: a case study

The first experimental case is a subset of an automotive system, showing how the addition of delays on the communication by selecting a late receiver instance (when oversampling) or sender instance (when undersampling) can be leveraged to find a feasible schedule.

The application configuration of Tables 2 and 3 is obtained from a prototypical X-by-Wire application from General Motors [3]. The application has 10 ECUs interconnected by a single FlexRay bus, executing 49 tasks, with periods of 1ms and 8ms respectively, and exchanging 132 signals. Tables 2 and 3 show periods and worst case execution time of tasks (in microseconds) and the size of each signal (in bits). The FlexRay bus is configured as follows: the application cycle is H= 8ms, the communication cycle l_{comm} = 1ms with n_{slot} = 22, and the slot size l_{slot} = 200bits, or 35μs. The system communication graph has several instances on communication with oversampling (from sender tasks with period 8ms to receiver tasks with period 1ms).

In the solution presented in [3] an approach to the problem that is typical of automotive system developers was used. In the definition of the software/message implementation of controls, the partial order of execution and flow preservation constraints are simply dropped and each task/message is only scheduled within the end of the corresponding period. This formally corresponds to the possible generation of variable communication delays between each sender and receiver and changes in the communication flows. This design is typically considered acceptable under the (often implicit) assumption that the controls implemented are tolerant with respect to this jitter (variable number of added delays). This practice, however, is challenged in those cases where safety-critical controls (such as X-by-wire) need to be *formally* demonstrated as correct. In the experiments presented in this paper we analyze in detail the possible addition of delays including the more stringent case where no added delay is allowed.

First, we try to find a feasible solution with no added delays, with an optimization function that relates to the extensibility

τ_i	e_i	T_i	C_i	τ_i	e_i	T_i	C_i
τ_8	e_9	8000	810	τ_{21}	e_5	1000	25
τ_9	e_9	8000	550	$\tau_{22}/\tau_{26}/\tau_{30}/\tau_{34}$	$e_5/e_6/e_7/e_8$	1000	60
τ_{11}	e_9	8000	100	$\tau_{23}/\tau_{27}/\tau_{31}/\tau_{35}$	$e_5/e_6/e_7/e_8$	1000	40
τ_{12}	e_9	8000	770	$\tau_{24}/\tau_{28}/\tau_{32}/\tau_{36}$	$e_5/e_6/e_7/e_8$	1000	20
τ_{13}	e_9	8000	200	$\tau_{25}/\tau_{29}/\tau_{33}$	$e_6/e_7/e_8$	1000	30
τ_{14}	e_9	8000	110	$\tau_{37}/\tau_{42}/\tau_{47}/\tau_{52}$	$e_1/e_2/e_3/e_4$	8000	1000
τ_{15}	e_9	8000	550	$\tau_{38}/\tau_{43}/\tau_{48}/\tau_{53}$	$e_1/e_2/e_3/e_4$	8000	500
τ_{16}	e_{10}	8000	780	$\tau_{39}/\tau_{44}/\tau_{49}/\tau_{54}$	$e_1/e_2/e_3/e_4$	8000	1500
τ_{10}	e_{10}	8000	510	$\tau_{40}/\tau_{45}/\tau_{50}/\tau_{55}$	$e_1/e_2/e_3/e_4$	8000	1300
τ_{17}	e_{10}	8000	190	$\tau_{41}/\tau_{46}/\tau_{51}/\tau_{56}$	$e_1/e_2/e_3/e_4$	8000	350
τ_{18}	e_{10}	8000	260	τ_{20}	e_{10}	8000	230
τ_{19}	e_{10}	8000	100				

Table 2: Tasks for the X-by-wire example

Signal	**Send**	**Size**	**Recv**	**Signal**	**Send**	**Size**	**Recv**
σ_1 to σ_4	τ_{15}	32	$\tau_{22}/\tau_{26}/\tau_{30}/\tau_{34}$	σ_{48}	τ_{22}	32	τ_{16}
σ_5 to σ_8	τ_{20}	32	$\tau_{22}/\tau_{26}/\tau_{30}/\tau_{34}$	σ_{49}	τ_{26}	32	τ_{16}
σ_9 to σ_{11}	τ_{23}	32	$\tau_{27}/\tau_{31}/\tau_{35}$	σ_{50} to σ_{53}	τ_{26}	16	τ_8/τ_{16}
σ_{12}, σ_{13}	τ_{21}	32	$\tau_{22}/\tau_{26}/\tau_{30}/\tau_{34}$	σ_{54} to σ_{63}	τ_{39}	16	τ_{12}
σ_{14}	τ_{12}	8	$\tau_{39}/\tau_{44}/\tau_{49}/\tau_{54}$	σ_{64}, σ_{65}	τ_{42}	16	τ_{12}
σ_{15} to σ_{18}	τ_{22}	16	τ_8/τ_{16}	σ_{66} to σ_{77}	τ_{44}	16	τ_{12}
σ_{19}, σ_{20}	τ_{23}	8	τ_8/τ_{16}	σ_{78} to σ_{87}	τ_{49}	16	τ_{12}
σ_{21} to σ_{23}	τ_{27}	32	$\tau_{23}/\tau_{31}/\tau_{35}$	σ_{88}, σ_{89}	τ_{52}	16	τ_{12}
σ_{24}, σ_{25}	τ_{25}	32	$\tau_{22}/\tau_{26}/\tau_{30}/\tau_{34}$	σ_{90} to σ_{99}	τ_{54}	16	τ_{12}
σ_{26} to σ_{32}	τ_{12}	16	$\tau_{39}/\tau_{44}/\tau_{49}/\tau_{54}$	σ_{100}	τ_8	1	τ_{17}
σ_{33}, σ_{34}	τ_{27}	8	τ_8/τ_{16}	σ_{101} to σ_{116}	τ_{12}	16	τ_{17}
σ_{35} to σ_{37}	τ_{31}	32	$\tau_{23}/\tau_{27}/\tau_{35}$	σ_{117} to σ_{124}	τ_{12}	16	τ_{17}/τ_{18}
σ_{38}, σ_{39}	τ_{29}	32	$\tau_{22}/\tau_{26}/\tau_{30}/\tau_{34}$	σ_{125}	τ_{30}	1	τ_8/τ_{16}
σ_{40}, σ_{41}	τ_{30}	16	τ_8/τ_{16}	σ_{126} to σ_{128}	τ_{34}	16	τ_8/τ_{16}
σ_{42} to σ_{44}	τ_{35}	32	$\tau_{23}/\tau_{27}/\tau_{31}$	σ_{129}, σ_{130}	τ_{37}	16	τ_{12}
σ_{45} to σ_{47}	τ_{33}	32	$\tau_{22}/\tau_{26}/\tau_{30}/\tau_{34}$	σ_{131}, σ_{132}	τ_{35}	8	τ_8/τ_{16}

Table 3: Signal list for the example

of the system: finding the schedule with the minimum number of used FlexRay slots. The problem is modeled in Python and solved using CPLEX 12.4 [57], on a machine with an Intel Core i5 2.4GHz CPU and 8GB memory. The solver very quickly, with a runtime of 0.1 second, determines that the problem has no feasible solution.

Then, the same problem is formulated by adding a 7ms communication delay on all the oversampling links with sender period 8ms and receiver period 1ms (the last instance of the receiver in the application cycle is actually reading the data). In this case, the solver finds a schedulable solution within 50 seconds, computing a feasible schedule that requires 44 slots out of the available 176 slots in each application cycle.

7.2. Computing the solution with minimum delay cost

Next, we use the same case study with artificially added signals and tasks to test the scalability, with the objective of finding the feasible solution that minimizes the weighted sum of functional delays. The delays are added to the communication links of controls that are sensitive to delays (a subset of the signals, as shown in Table 4). In this case study, the weights applied to delays are not obtained from a simulation stage (the original Simulink models were not available), but are randomly assigned under the assumption that the solver time and the possibility of finding an optimal solution are not affected by the actual delay costs. To support this assumption, we choose another random weight configuration different from Table 4, with values 2, 3, 4, 1, 1, 2, 3, 2, 2, 4, 3, 3, 1, and 2 respectively. We compare the two configurations using the two-step approach (which

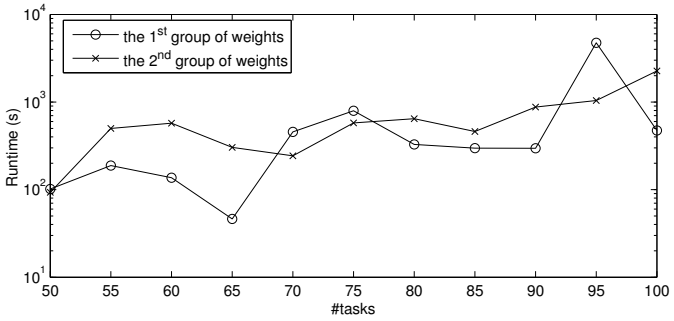


Figure 12: Comparison of runtime of problems with different weight values.

Signal	σ_1	σ_6	σ_{21}	σ_{24}	σ_{39}	σ_{40}	σ_{41}
Weight	1.2	1.1	0.8	0.9	2	0.5	1
Signal	σ_{42}	σ_{43}	σ_{44}	σ_{63}	σ_{76}	σ_{109}	σ_{125}
Weight	1.5	2.4	0.8	1.2	1	0.9	1

Table 4: Critical signals and their weights

is detailed in the rest of the subsection). Under both configurations, the optimal solutions are found in a short amount of time, as shown in Figure 12. We can see that the runtime is similar for the two groups of weights.

The MILP formulation may not be scalable to large designs. We consider a complete MILP formulation and a heuristic based on divide-and-conquer. In the first approach, we formulate all variables, constraints and the objective into a single MILP formulation. We call it a *one-step approach*. For the original design problem (Tables 2 and 3), the CPLEX solver finds the optimal solution in 25131 seconds (or about 7 hours). However, the large runtime for a problem of relatively small size is a clear indication that this formulation can hardly scale up to problems of larger size (confirmed by the results of our scalability analysis).

To simplify the problem formulation and reduce the runtime of the solver, we propose a *two-step approach*. The first step aims at the reduction in the number of signals by applying a simple bin-packing algorithm. Signals that have the same sender task and the same period are packed and mapped into a FlexRay message (a FlexRay static slot) as an atomic unit. The second step is the MILP formulation of the optimization problem to schedule the tasks and the messages. After the first stage, the number of messages to be scheduled on FlexRay is now significantly reduced and the MILP problem becomes much easier to solve. Using this approach, the optimal solution for our case study configuration can be found in just 151 seconds. In this case, the two-step approach gives the same optimal result as the one-step approach.

Next, we study the scalability and quality of our proposed two-step approach by adding a set of randomly generated signals and tasks. In the first series of experiments, the number of tasks is fixed at 49 and system configurations of increasing complexity are generated by adding 16 signals in each new experiment, from 132 to a maximum of 292. Each configuration is solved using the two-step approach, and the results and run-

time are shown in Figures 13a and 14a. The graphs show that despite the added number of signals, the cost of the solution that is found after optimization does not increase significantly. This is probably because the FlexRay schedule allows to accommodate new signals in the existing slots quite easily at first and only requires shifting signals to later slots when the bus is approaching high utilizations. However, the runtime of the algorithm is significantly affected by the communication complexity. For our system configuration with 132 signals, the solver needs 151 seconds to find the optimal solution, but the execution time increases to 6626 seconds for the 292-signal case.

In the second set of experiments, we randomly increase the number of tasks from 50 to 100 while keeping the number of signals unchanged. Figures 13b and 14b show the result and runtime of the experiments. The number of tasks (an indication of the computation complexity) is now affecting the cost value of the computed optimum much more than in the scalability analysis on number of signals. This is because adding more tasks increases their response times which reduces the window of scheduling. Thus it has a larger effect on schedulability and the need to add functional delay.

For the task scalability analysis, we compare the execution times and the cost of the computed solutions for the one-step approach and the simplified two-step approach. As shown in Figure 13c, 6 configurations with different numbers of tasks are considered. For the first 4 cases, both methods compute the true optimum solutions (the solutions computed by the two methods not only have the same cost but are exactly the same). For the last 2 cases, the one-step approach cannot find the optimal solution as the solver runs out of memory. When this happens, the best solutions are found by the solver with gaps 4% and 1.8% respectively (the gap is an upper bound of the distance to the global optimum). The suboptimal solution (with value 40) computed by the one-step approach on the fifth case (with 90 tasks) is actually worse than the one (with value 39.2) found using the two-step approach. Since the gap for the one-step solution is 4% and the solution found by the two-step approach is 2% better, it is also (in the worst case) within 2% of the global optimum. For the sixth case (with 100 tasks), the solution found using the one-step optimization (with value 44) outperforms the one computed with the two-step algorithm (with value 44.8). However, even if we consider the lower bound on the cost of the optimal solution $44 \times (1 - 1.8\%) = 43.2$ (which is not likely to be the global optimum), the solution computed by the two-step approach is still within 4%. Figure 14c shows a comparison of the runtimes using the two approaches. It can be seen that the execution time of the two-step heuristic is typically two magnitudes less than the one of the one-step approach.

7.3. Case study: active safety functions

Finally, we apply the one-step optimization formulation to another case study available in the literature [44], a set of active safety functions including a vehicle adaptive cruise controller (ACC), an electric power steering (EPS), and traction control (TC) functions. The case study is relatively small, with 28 tasks and 31 signals (the details can be found in [44]). The selected signals and their weights are listed in Table 5. As for [3], our method cannot be directly compared to the solution in [44]. In the definition of the scheduling, the method in [44] is only interested in finding any schedulable solution (with arbitrary and possibly non-deterministic communication delays), while we are interested in *the solution with the smallest cost in terms of deterministic additional delays*.

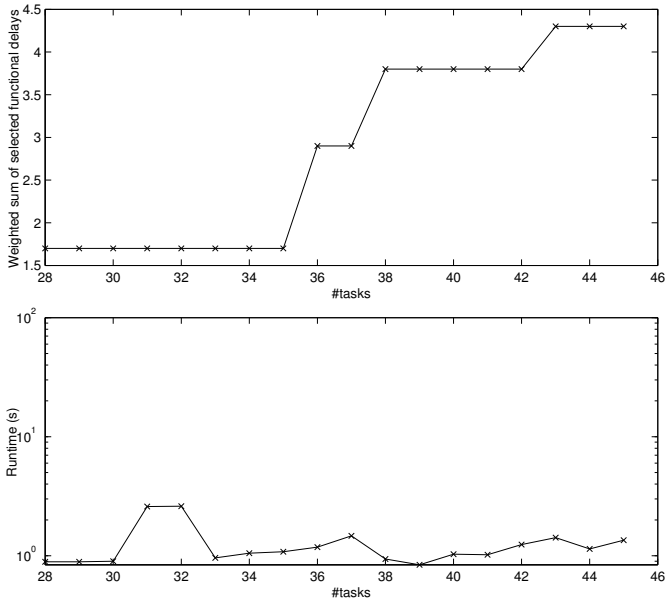


Figure 15: Weighted sum of selected functional delays and runtime of the optimization method for the active safety functions.

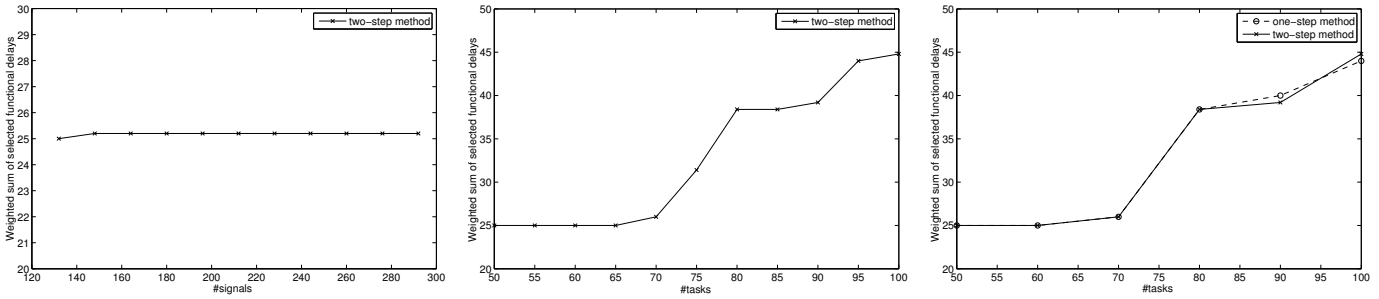
Signal	σ_1	σ_2	σ_3	σ_9	σ_{12}	σ_{13}	σ_{16}
Weight	1	1.5	1.2	1.1	2.4	0.8	0.8
Signal	σ_{18}	σ_{19}	σ_{23}	σ_{26}	σ_{28}		
Weight	1.2	1	0.9	2	0.5		

Table 5: Critical signals and their weights for the case study on active safety functions [44].

After applying the one-step optimization approach, the optimum solution is found, requiring the addition of functional delays to only 2 signals (1 delay unit on each link). We also perform a scalability analysis by increasing the number of tasks from 28 to 45. The weighted sum of selected functional delays and the runtime of the algorithm are shown in Figure 15. The one-step approach returns the optimal results for each of the configurations in no more than 3 seconds.

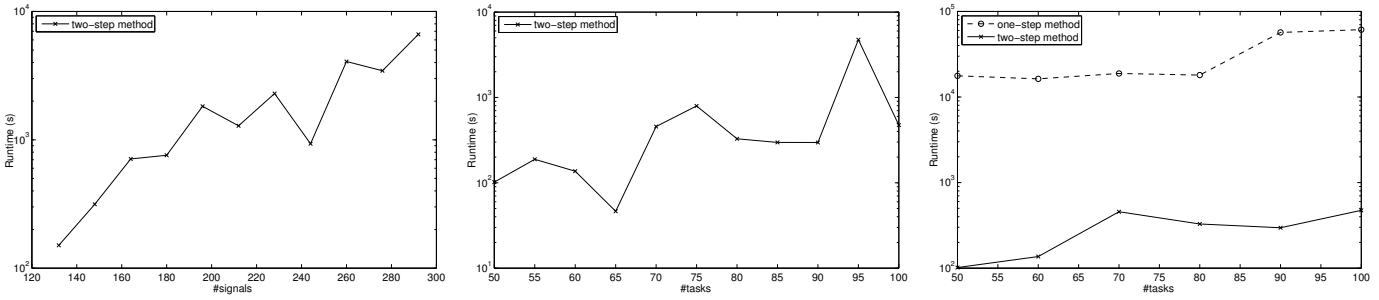
8. Conclusions

FlexRay offers the possibility of a deterministic communication and can be used to define distributed implementations that are provably equivalent to synchronous reactive models like those created from Simulink. However, the low level communication layers and the FlexRay schedule must be carefully designed to ensure the preservation of model semantics, especially when models include subsystems executing at different rates. In this paper we provided a discussion and an analysis of



(a) Functional delays vs. number of signals (b) Functional delays vs. number of tasks (c) Comparison of one-step and two-step approaches on functional delay

Figure 13: Weighted sum of selected functional delays.



(a) Runtime vs. number of signals (b) Runtime vs. number of tasks (c) Comparison of one-step and two-step approaches on runtime

Figure 14: Runtime of the optimization methods.

the issues that need to be faced when defining a FlexRay communication schedule in a model-based design flow, where tasks and messages provide a distributed implementation of a synchronous model. We provide a formulation of an optimization problem that computes the optimal solution with respect to the number of additional delays when a flow-preserving implementation is not possible. The aforementioned scheduling options are applied to an X-by-wire system and an active-safety case study to highlight tradeoffs between schedulability and additional functional delays.

References

[1] FlexRay Consortium. FlexRay Communications System Protocol Specification, Version 3.0.1. *Available at http://www.flexray.com*, 2010.

[2] M. Grenier, L. Havet, and N. Navet. Configuring the communication on flexray: the case of the static segment. *Euromicro Conference on Real-Time Systems*, 2008.

[3] H. Zeng, M. Di Natale, A. Ghosal, and A. Sangiovanni-Vincentelli. Schedule Optimization of Time-Triggered Systems Communicating Over the FlexRay Static Segment. *IEEE Transactions on Industrial Informatics*, 7(1):1–17, February 2011.

[4] M. Lukasiewycz, R. Schneider, D. Goswami, and S. Chakraborty. Modular Scheduling of Distributed Heterogeneous Time-triggered Automotive Systems. *Asia and South Pacific Design Automation Conference*, 2012.

[5] M. Lukasiewycz, M. Glaß, J. Teich, and P. Milbredt. Flexray schedule optimization of the static segment. *IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, 2009.

[6] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, January 2003.

[7] *SCADE Suite Product*, web page: http://www.esterel-technologies.com/products/scade-suite (Retrieved on Dec. 31, 2012).

[8] *The Mathworks Simulink and StateFlow Users Manuals*, Mathworks, web page: http://www.mathworks.com (Retrieved on Dec. 31, 2012).

[9] *Prover Technology*, web page: http://www.prover.com (Retrieved on Dec. 31, 2012).

[10] H. Kopetz and G. Bauer. The Time-Triggered Architecture. *Proceedings of the IEEE*, 91(1):112–126, January 2003.

[11] P. Caspi, A. Curic, A.Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications. *ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, 2003.

[12] M. Di Natale and H. Zeng. Time Determinism and Semantics Preservation in the Implementation of Distributed Functions over FlexRay. *Society of Automotive Engineers World Congress*, 2010.

[13] D. Potop-Butucaru, B. Caillaud, and A. Benveniste. Concurrency in synchronous systems. *Formal Methods in System Design*, 28(2):111–130, March 2006.

[14] P. Caspi and A. Benveniste. Time-robust discrete control over networked loosely time-triggered architectures. *IEEE Control and Decision Conference*, 2008.

[15] P. Caspi and M. Pouzet. Synchronous kahn networks. *ACM SIGPLAN Conference on Functional Programming*, 1996.

[16] S. Tripakis, C. Pinello, A. Benveniste, A. Sangiovanni-Vincentelli, P. Caspi and M. Di Natale. Implementing Synchronous models on Loosely Time-Triggered Architectures. *IEEE Transactions on Computers*, 57(10):1300–1314, October 2008.

[17] A. Jantsch and I. Sander. Models of Computation and Languages for Embedded System Design. *IEEE Proceedings on Computers and Digital Techniques*, 152(2):114–129, March 2005.

[18] E. Lee and A. Sangiovanni-Vincentelli. A Framework for Comparing

Models of Computation. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, December 1998.

[19] F. Boussinot and R. de Simone. The Esterel language. in *Proceedings of the IEEE*, vol. 79, pp. 1293–1304, Sept. 1991.

[20] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A declarative language for programming synchronous systems. in *ACM Symp. Principles Program. Lang. (POPL)*, Munich, 1987, pp. 178–188.

[21] J. Forget, F. Boniol, D. Lesens, and C. Pagetti. A multi-periodic synchronous data-flow language. In *11th IEEE High Assurance Systems Engineering Symposium (HASE'08)*, Nanjing, Dec. 2008.

[22] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. in *Sci. Comput. Program*, vol. 19, pp. 87–152, Nov. 1992.

[23] S. A. Edwards. An Esterel compiler for large control-dominated systems. in *IEEE Trans. Computer-Aided Design*, vol. 21, Feb. 2002.

[24] D. Weil, V. Berlin, E. Closse, M. Poize, P. Venier, and J. Pulou. Efficient compilation of Esterel for real-time embedded systems. in *Proc. Int. Conf. Compilers, Architecture, and Synthesis for Embedded Syst.*, San Jose, CA, Nov. 2000, pp. 2–8.

[25] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. in *Proceedings of the IEEE*, 1987, 75, (9), pp. 1235–1245

[26] E.A. Lee and D.G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. in *IEEE Transactions on Computers*, 36(1):24–35, 1987.

[27] S. Bhattacharyya, E. Lee, and P. Murthy. *Software Synthesis from Dataflow Graphs*. Kluwer, 1996.

[28] Falk H., Marwedel P. Source Code Optimization Techniques For Data Flow Dominated Embedded Software. *Springer*, Berlin 2004

[29] J. Falk, J. Keinert, C. Haubelt, J. Teich, and S. Bhattacharyya. A generalized static data flow clustering algorithm for mpsoc scheduling of multimedia applications. *ACM International Conference on Embedded software*, 2008.

[30] Strehl, K., Thiele, L., Gries, M., Ziegenbein, D., Ernst, R., and Teich, J. FunState - an internal design representation for codesign. *IEEE Transactions on Very Large Scale Integration* (VLSI), 2001, 9(4), pp. 524–544

[31] Hoare, C.A.R. Communicating sequential processes. *Communications of the ACM*, 1978, 21, (8), pp. 666–676

[32] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[33] Andrzej Wasowski. On Efficient Program Synthesis from Statecharts. *Proc. ACM SIGPLAN Conf. of Languages, Compilers, and Tools for Embedded Systems*, 2003.

[34] Plishker, William and Sane, Nimish and Bhattacharyya, Shuvra S. A generalized scheduling approach for dynamic dataflow applications. *Proceedings of the DATE Conference* , 2009, Nice, France

[35] Bhattacharyya, S. S. and Murthy, P. K. and Lee, E. A. Optimized software synthesis for synchronous dataflow. *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors*, 1997, Washington, DC, USA.

[36] Li, M., Wang, H., and Li, P. Tasks mapping in multicore based system: Hybrid ACO&GA approach. *Proceedings of the International Conference on ASIC, 2003*.

[37] F. Reimann, M. Glaß, C. Haubelt, M. Eberl, and J. Teich. Improving platform-based system synthesis by satisfiability modulo theories solving. *IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, 2010.

[38] R. Niemann, P. Marwedel. An Algorithm for Hardware/Software Partitioning Using Mixed Integer Linear Programming. *Design Automation for Embedded Systems* March 1997, Vol. 2 (2), pp 165-193

[39] P. Pop, P. Eles, and Z. Peng. Schedulability-Driven Communication Synthesis for Time Triggered Embedded Systems. *Real-Time Systems*, 26(3):297–325, April 2004.

[40] A. Hamann and R. Ernst. TDMA time slot and turn optimization with evolutionary search techniques. *Conference on Design, Automation and Test in Europe*, 2005.

[41] K. Schmidt and E. G. Schmidt. Message scheduling for the flexray protocol: The static segment. *IEEE Transactions on Vehicular Technology*, 58(5):2170–2179, June 2008.

[42] B. Tanasa, U. Bordoloi, P. Eles, and Z. Peng. Scheduling for Fault-Tolerant Communication on the Static Segment of FlexRay. *IEEE Real-Time Systems Symposium*, 2010.

[43] B. Tanasa, U. Bordoloi, P. Eles, and Z. Peng. Reliability-Aware Frame Packing for the Static Segment of FlexRay. *ACM International Conference on Embedded software*, 2011.

[44] S. Ding, N. Murakami, H. Tomiyama, and H. Takada. A ga-based scheduling method for flexray systems. *ACM International Conference on Embedded software*, 2005.

[45] T. Pop, P. Pop, P. Eles, Z. Peng, and A. Andrei. Timing analysis of the FlexRay communication protocol. *Real-Time Systems*, 39(1-3):205–235, August 2008.

[46] R. Racu, M. Jersak, and R. Ernst. Applying sensitivity analysis in real-time distributed systems. *IEEE Real Time and Embedded Technology and Applications Symposium*, 2005.

[47] W. Zheng, Q. Zhu, M. Di Natale, and A. Sangiovanni-Vincentelli. Definition of task allocation and priority assignment in hard real-time distributed systems. *IEEE Real-Time Systems Symposium*, 2007.

[48] Q. Zhu, H. Zeng, W. Zheng, M. Di Natale, and A. Sangiovanni-Vincentelli. Optimization of task allocation and priority assignment in hard real-time distributed systems. *ACM Transactions on Embedded Computing Systems*, 11(4):85, December 2012.

[49] T. Pop, P. Eles, and Z. Peng. Design optimization of mixed time/event-triggered distributed embedded systems. *IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, 2003.

[50] S. Samii, A. Cervin, P. Eles, and Z. Peng. Integrated scheduling and synthesis of control applications on distributed embedded systems. *Conference on Design, Automation and Test in Europe*, 2009.

[51] A. Metzner and C. Herde. RTSAT– an optimal and efficient approach to the task allocation problem in distributed architectures. *IEEE Real-Time Systems Symposium*, 2006.

[52] P. Emberson and I. Bate. Stressing search with scenarios for flexible solutions to real-time task allocation problems. *IEEE Transactions on Software Engineering*, 36(5):704–718, September 2010.

[53] A. Hamann, R. Racu, and R. Ernst. Methods for multi-dimensional robustness optimization in complex embedded systems. *ACM International Conference on Embedded software*, 2007.

[54] J. P. Vielma, S. Ahmed, and G. Nemhauser. Mixed-integer models for non-separable piecewise linear optimization: unifying framework and extensions. *Operations Research*, 58:303–315, 2010.

[55] OSEK/VDX operating system specification version 2.2.3. http://www.osek-vdx.org, February 2005.

[56] G. Wang, M. Di Natale, and A. Sangiovanni-Vincentelli, Optimal synthesis of communication procedures in real-time synchronous reactive models. *IEEE Transactions on Industrial Informatics*, 6(4):729–743, November 2010.

[57] *CPLEX Optimizer*, web page: http://www.ibm.com/software/commerce/optimization/cplex-optimizer/ (Retrieved on Dec. 31, 2012).